Desarrollo a nivel de servidor

Capa de negocio/aplicación

Luis Valencia Cabrera (lvalencia@us.es)

Research Group on Natural Computing
Departamento de Ciencias de la Computación e Inteligencia Artificial
Universidad de Sevilla

15-04-2026, Bases de Datos

Índice

1. Introducción
2. Entorno de Trabajo
3. Creación de Servidor
4. API servidor sobre MongoDB
5. API servidor sobre MySQL

El Servidor: El guardián de la lógica y el dato

Si el navegador (cliente) es la interfaz, el servidor es el **motor**. Sus funciones principales son:

- **Gestión de Datos:** Es el único que tiene "la llave"para hablar con la Base de Datos de forma segura.
- **Persistencia de Archivos:** Almacena imágenes, PDFs y documentos que el usuario sube.
- **Seguridad:** Centraliza la validación (quién puede ver qué) para evitar que el cliente manipule los datos.
- **Reglas de Negocio:** Realiza los cálculos y procesos que no queremos que el usuario vea o modifique.

Sin servidor, la web sería una revista estática; con servidor, es una aplicación viva.

Tecnologías de Back-end: Muchas lenguas para un fin

No existe una única forma de programar un servidor. Según el proyecto:

- **PHP:** El clásico de la web (WordPress, Laravel).
- **Python:** Muy potente en ciencia de datos (Django, Flask).
- **Java / C#:** Estándares para grandes corporaciones (Spring, .NET).
- **Node.js:** La revolución del JavaScript asíncrono.

¿Por qué elegimos Node.js en esta asignatura?

1. **Sinergia:** Ya conocéis la sintaxis. Aprovechamos vuestro esfuerzo en aprender JS.
2. **Rendimiento:** Su arquitectura está diseñada para gestionar miles de conexiones simultáneas.
3. **Mercado:** Es la tecnología con mayor crecimiento y demanda en el ecosistema de las Startups y el Cloud moderno.

Node.js: De juguete a estándar industrial

¿Por qué empresas como **Netflix, LinkedIn, Uber o PayPal** migraron gran parte de su infraestructura a Node?

- **Unificación de equipos:** Los mismos desarrolladores pueden tocar el Front-end y el Back-end (Full-Stack).
- **Velocidad de desarrollo:** Enorme comunidad y repositorio de bibliotecas.
- **Eficiencia:** Consume menos recursos de servidor para entrada/salida de datos (perfecto para nuestras futuras APIs[1]).

"Learn once, write everywhere."

[1] **API (Application Programming Interface):** Conjunto de servicios que el servidor **expone** al exterior. Define qué funciones puede invocar el cliente (ej: `GET /pacientes`) y qué datos recibirá a cambio.

NPM: El mayor registro de software del mundo

Node.js se apoya en **NPM** (Node Package Manager), el ecosistema de módulos más grande que existe.

- **Escala:** Cuenta con más de 2.5 millones de **bibliotecas*** disponibles.
- **Crecimiento:** Supera en volumen a los repositorios de Java (Maven) o Python (PyPI).
- **Comunidad:** Casi cualquier problema técnico (manejo de fechas, validación de emails, conexión a BD) ya tiene una solución publicada.

Nota terminológica

*Aunque veréis mucho "librería" por influencia del inglés (*library*), el término correcto en español es **biblioteca** de software.

Índice

1 Introducción
2 Entorno de Trabajo
3 Creación de Servidor
4 API servidor sobre MongoDB
5 API servidor sobre MySQL

Node.js: JavaScript en el Sistema Operativo

En el cliente, JavaScript vive "enjaulado" en el navegador. En el servidor, con **Node.js**[2], le damos acceso total a la máquina.

- **Motor V8:** El mismo motor de alto rendimiento que usa Chrome, pero ejecutándose de forma independiente.
- **Acceso al Sistema:** Permite leer/escribir archivos, gestionar redes y conectar con Bases de Datos.
- **Modelo Asíncrono:** Basado en un bucle de eventos (*Event Loop*) que permite gestionar múltiples tareas sin bloquear el hilo principal.

Comprobación de instalación

En la terminal:

> `node -v` (Versión del entorno)

> `npm -v` (Versión del gestor de paquetes)

[2] Que podemos descargar de `https://nodejs.org/es/download` e instalar fácilmente

NVM: ¿Por qué gestionar versiones?

En el mundo real, distintos proyectos pueden requerir distintas versiones de Node.js. **NVM** es la herramienta que nos permite saltar de una a otra sin desinstalar nada.

- **El problema:** Instalar Node directamente desde la web oficial nos "ancla" a una sola versión.
- **La solución:** NVM instala múltiples versiones en carpetas aisladas y nos permite elegir cuál usar con un comando.

Instalación

- **Mac / Linux:** github.com/nvm-sh/nvm
- **Windows (nvm-windows):** github.com/coreybutler/nvm-windows

Trabajando con NVM: Comandos básicos

Una vez instalado, NVM se gestiona totalmente desde la terminal. Estos son los comandos que más utilizarás:

- **Instalar una versión específica:**

```
nvm install 24.14.1  # Instala la versión LTS actual
```

- **Ver qué versiones tienes instaladas:**

```
nvm list             # En Windows
nvm ls               # En Mac/Linux
```

- **Cambiar la versión activa:**

```
nvm use 20.11.0  # Cambia el entorno a otra versión
```

- **Saber qué versión estás usando ahora:**

```
node -v   # Debería coincidir con la elegida en NVM
```

Node y npm: El flujo de trabajo de un proyecto

Todo proyecto de Node comienza con el archivo fundamental: **package.json** (configuración del proyecto, dependencias, scripts...)

Tras crear la carpeta donde queramos el proyecto y movernos a ella:

1. **Inicialización:** `npm init -y` (crea esqueleto base en package.json).
2. **Instalación de paquetes:** `npm install express`
3. **Gestión de dependencias:**
   - `dependencies`: bibliotecas necesarias para que el código funcione.
   - `node_modules`: carpeta (gigante) donde se descargan físicamente las librerías.

Regla de Oro

NUNCA se sube la carpeta `node_modules` a repositorios de código. Se usa un archivo `.gitignore` para evitarlo.

Hola Mundo en el Servidor

Crea un archivo llamado `app.js`:

```
console.log("Hola desde el servidor");
const os = require('os'); // Módulo nativo
console.log("Memoria libre:", os.freemem() / 1024**3, "GB");
```

¿Cómo lo ejecutamos?

- En la terminal: `node app.js`
- (Opcional) Para desarrollo, es habitual usar, en lugar de la llamada a `node`, a `nodemon`, que vigila cambios en archivos y reinicia Node.js automáticamente al guardar cambios. `nodemon app.js`[3]

[3] Podemos instalarlo globalmente mediante: `npm install -g nodemon`, o bien localmente como dependencia de desarrollo (no de producción) mediante: `npm install --save-dev nodemon`

Módulos: Exportar e Importar

Node.js ha evolucionado. Verás dos formas de importar código:

CommonJS (Tradicional)

```
const express = require('express');
module.exports = miFuncion;
```

ES Modules (Moderno)

```
import express from 'express';
export default miFuncion;
```

*Para usar la sintaxis moderna (**import**), debemos añadir `"type": "module"` en nuestro `package.json`.*

Índice

1 Introducción
2 Entorno de Trabajo
3 Creación de Servidor
4 API servidor sobre MongoDB
5 API servidor sobre MySQL

¿Qué requiere crear un servidor web?

Elementos fundamentales:

- `Puerto`: canal de comunicación (ej. 3000, 8080)
- `Rutas`: URLs que responde (ej. /, /usuarios)
- `Métodos HTTP`: GET (leer), POST (crear), PUT, DELETE
- `Respuesta`: HTML, JSON, texto, etc.

Dos enfoques en Node.js:

`http.createServer()` (nativo)	Verboso, rutas manuales
Express (framework)	Simple, estándar actual (>90 % proyectos)

Instalación de express: `npm install express`

Express: El estándar para APIs

Express simplifica la creación de rutas y la escucha de peticiones.

```
import express from 'express';
const app = express();
const port = 3000;

// Ruta principal (Home)
app.get('/', (req, res) => {
  res.status(200).send('¡Bienvenid@ a su web!');
});

app.listen(port, () => {
  console.log(`Servidor escuchando en http://localhost:${port}`);
});
```

Como en el ejemplo de script básico, lo lanzaríamos con `node archivo.js` o `nodemon archivo.js`

Explicación del código Express

- `import express from 'express'` → importar la librería
- `const app = express()` → crear la aplicación
- `const port = 3000` → definir puerto (8080, 5000...)
- `app.get('/', f)` → ruta en la raíz del servidor esperando petición de tipo GET
- `res.send('...')` → enviar respuesta al cliente
- `app.listen(port, ...)` → arrancar servidor en el puerto

Probar: Abrir navegador en `http://localhost:3000`

API REST: Conceptos básicos

API (Application Programming Interface): Permite que aplicaciones se comuniquen entre sí.

Principios REST:

- Recursos identificados por `URL` (ej. `/usuarios`, `/productos`)
- Métodos HTTP para operar sobre recursos
- Sin estado (*stateless*): cada petición contiene toda la información necesaria
- Respuestas en formato estándar (JSON, XML, etc.)

Ejemplo: Tienda online

- `GET /productos` → listar productos
- `POST /productos` → crear nuevo producto
- `GET /productos/7` → ver producto con ID 7

Diseño de URLs en API REST

Estructura básica: `http://api.ejemplo.com/v1/recurso`

Método	URL	Qué hace
GET	`/usuarios`	Listar todos
GET	`/usuarios/5`	Obtener usuario ID 5
POST	`/usuarios`	Crear nuevo usuario
PUT	`/usuarios/5`	Actualizar usuario ID 5
DELETE	`/usuarios/5`	Eliminar usuario ID 5

Buenas prácticas:

- Usar **sustantivos en plural**: `/productos` (no `/getProductos`)
- **Anidados** para relaciones: `/usuarios/5/pedidos`
- Versiones en la URL: `/v1/`, `/v2/`

Middleware: El intermediario

Un **Middleware** es una función que tiene acceso al objeto de petición (`req`), al de respuesta (`res`) y a la siguiente función de la cadena (`next`).

- **¿Cómo funciona?** Actúa como una aduana: revisa la petición, la transforma o la bloquea si es necesario.
- **La tubería (Pipeline):** Las peticiones atraviesan los middlewares en el orden en que se definen en el código.

Funciones comunes

- **Parsing:** Traducir el cuerpo de la petición (JSON).
- **Seguridad:** Comprobar si el usuario está logueado.
- **Logs:** Registrar quién entra y qué pide.

Configuración esencial

Para que nuestra API sea funcional y segura, necesitamos configurar dos middlewares clave antes de nuestras rutas:

1. **express.json():** Por defecto, Node no sabe leer el cuerpo de una petición POST. Este middleware "traduce" el JSON entrante y lo coloca en `req.body`.
2. **CORS (Cross-Origin Resource Sharing):** Por seguridad, los navegadores bloquean peticiones entre distintos dominios. Este middleware permite que tu Front-end (ej. puerto 5173) hable con tu Back-end (puerto 3000).

```
import express from 'express';
import cors from 'cors';

const app = express();

app.use(cors());           // Permitir peticiones externas
app.use(express.json());   // Habilitar lectura de JSON
```

¿Qué status debemos devolver?

Aunque Express asigna el **200** por defecto, como desarrolladores debemos ser precisos:

- **200 OK:** Todo correcto (por defecto).
- **201 Created:** Recurso creado con éxito (típico de `POST`).
- **204 No Content:** Éxito, pero no devuelve datos (típico `DELETE`).
- **400 Bad Request:** El cliente envió datos mal formados.
- **401 Unauthorized:** Falta autenticación.
- **404 Not Found:** El recurso no existe.
- **500 Internal Server Error:** Error de nuestro código (bug).

Regla de oro

"Usa siempre el código más específico posible. Ayuda al desarrollador del Front-end a saber qué ha fallado sin leer el cuerpo del mensaje."

Tipos de parámetros en la petición

Existen tres formas principales en las que el cliente envía información al servidor:

1. **Route Params (req.params):** Variables incrustadas en la URL para identificar un recurso concreto.
   - Ejemplo: `/api/estudiantes/:id` → `/api/estudiantes/123`
2. **Query Params (req.query):** Parámetros opcionales al final de la URL para filtrar o buscar.
   - Ejemplo: `/api/estudiantes?curso=3&grado=salud`
3. **Body (req.body):** Datos complejos (objetos JSON) enviados de forma "oculta" en la petición.
   - Se usa en **POST** y **PUT**. Requiere el middleware `express.json()`.

Identificación de Recursos: req.params

Los parámetros de ruta (`req.params`) se definen con :

```
app.get('/paciente/:id', (req, res) => {
  // URL: /paciente/45
  const { id } = req.params;
  res.send(`Ficha del paciente: ${id}`);
});
```

```
app.get('/archivo/:anio/:mes/:dia', (req, res) => {
  // URL: /archivo/2026/04/27
  const { anio, mes, dia } = req.params;
  res.json({ fecha: `${dia}/${mes}/${anio}` });
});
```

```
app.get('/hospital/:hospId/planta/:pId/cama/:cId', (req, res) => {
  // URL: /hospital/H-Valme/planta/3/cama/12
  const { hospId, pId, cId } = req.params;
  res.json({ hospital: hospId, planta: pId, cama: cId });
});
```

Filtrado y Búsqueda: req.query

Los **Query Parameters** modifican la respuesta sin alterar la jerarquía de la URL. Ideales para filtros opcionales.

Caso 1: Filtro Simple - *URL: /buscar?medico=garcia*

```
app.get('/buscar', (req, res) => {
  const nombre = req.query.medico; // "garcia"
  res.send("Buscando resultados para el Dr. " + nombre);
});
```

Caso 2: Múltiples Criterios - *URL: /analisis?prioridad=alta&tipo=glucosa*

```
app.get('/analisis', (req, res) => {
  const prioridad = req.query.prioridad; // "alta"
  const prueba = req.query.tipo;         // "glucosa"

  res.json({ urgencia: prioridad, tipo_analisis: prueba });
});
```

Envío de Datos Estructurados: req.body

El **Request Body** permite enviar información compleja y privada. Es el estándar para crear (POST) o actualizar recursos.

Ejemplo de Registro (POST)

Datos enviados: { "nombre": "Ana", "edad": 30 }

```
app.post('/pacientes', (req, res) => {
  const nombrePaciente = req.body.nombre;
  const edadPaciente = req.body.edad;

  res.status(201).json({
    mensaje: "Paciente registrado con éxito",
    usuario: nombrePaciente
  });
});
```

Paso de Arrays: ¿Query o Body?

Por Query: Ideal para filtros rápidos. URL: /buscar?id=1&id=2

```
app.get('/buscar', (req, res) => {
  const ids = req.query.id; // ["1", "2"]
  res.send(JSON.stringify(ids.map((n) => parseInt(n))));
});
```

Por Body: Ideal para procesos masivos o datos complejos.

```
// Cliente envía JSON: { "ids": [1, 2, 3, 4, 5...] }
app.post('/procesar', (req, res) => {
  console.log(req.body.ids.length);
});
```

Consejo

Si la lista de elementos es muy larga, usa siempre **POST** y **Body** para evitar que la URL se corte o devuelva error de "URL Too Long".

JavaScript Moderno: Desestructuración

La desestructuración es una sintaxis que nos permite "extraer" valores de objetos o arrays de forma rápida y limpia. Puede aplicarse a objetos:

```
// Forma antigua
const nombre = req.body.nombre;
const edad = req.body.edad;

// Con desestructuración
const { nombre, edad } = req.body;
```

y a arrays:

```
const [primero, segundo] = ["React", "Node"];
// primero = "React", segundo = "Node"
```

¿Por qué lo usamos?

Evita repetir `req.body`, `req.query` o `req.params` constantemente y hace el código mucho más legible.

Índice

1. Introducción
2. Entorno de Trabajo
3. Creación de Servidor
4. API servidor sobre MongoDB
5. API servidor sobre MySQL

Configuración y Conexión a MongoDB[4]

Sintaxis CommonJS

```
const { MongoClient, ObjectId } = require("mongodb");
```

Sintaxis ES Modules

```
import { MongoClient, ObjectId } from "mongodb";
```

```
const url = "mongodb://localhost:27017";
const client = new MongoClient(url);
const dbName = "ejercicios";
let db;

async function connectDB() {
  try {
    await client.connect();
    console.log("Conectado a MongoDB OK");
    db = client.db(dbName);
  } catch (e) {
    console.error("Error de conexión:", e);
  }
}
connectDB();
```

[4] Instalación: npm install mongodb

API de Estudiantes: Lectura y Creación

```
// GET: Obtener todos los estudiantes
app.get('/api/estudiantes', async (req, res) => {
  const lista = await db.collection('estudiantes').find().toArray();
  res.json(lista);
});

// POST: Registrar un nuevo estudiante
app.post('/api/estudiantes', async (req, res) => {
  const nuevo = req.body; // El middleware express.json() hace su trabajo
  if (!nuevo.nombre || !nuevo.expediente) {
    return res.status(400).json({ error: "Datos incompletos" });
  }
  const resultado = await db.collection('estudiantes').insertOne(nuevo);
  res.status(201).json(resultado);
});
```

El código de estado **201** indica que un recurso ha sido creado con éxito, mientras que el **400** indica un error del cliente.

API de Estudiantes: Edición y Borrado

```
// PUT: Actualizar datos de un estudiante por su ID
app.put('/api/estudiantes/:id', async (req, res) => {
  const id = req.params.id;
  const datosActualizados = req.body;
  await db.collection('estudiantes').updateOne(
    { _id: new ObjectId(id) },
    { $set: datosActualizados }
  );
  res.json({ mensaje: "Estudiante actualizado" });
});

// DELETE: Eliminar un estudiante
app.delete('/api/estudiantes/:id', async (req, res) => {
  const id = req.params.id;
  await db.collection('estudiantes').deleteOne({ _id: new ObjectId(id) });
  res.status(204).send(); // 204: No Content (borrado con éxito)
});
```

Sincronización: El Cliente envía, el Servidor recibe

Para que la comunicación funcione, el **Fetch** (Cliente) debe coincidir con el **req** (Servidor).

T	Cliente (Fetch)	Servidor (Express)
P	`fetch('/api/estudiantes/123')`	`app.get('/.../:id')` → `req.params.id`
Q	`fetch('/api/estudiantes?nota=5')`	`app.get('/...')` → `req.query.nota`
B	`fetch('/...', {method:'POST',..})`	`app.post('/...')` → `req.body`

```
// CLIENTE: Pide aprobados
fetch('/api/estudiantes?estado=aprobado');

// SERVIDOR: Filtra y responde
app.get('/api/estudiantes', async (req, res) => {
  const filtro = req.query.estado; // "aprobado"
  const resultados =
    await db.collection('estudiantes').find({estado:filtro}).toArray();
  res.json(resultados);
});
```

Enviando datos complejos (POST/PUT)

Cuando usamos el **body**, el cliente debe cumplir tres requisitos para que el servidor lo entienda:

```
// Lado del CLIENTE
fetch('/api/estudiantes', {
  method: 'POST',
  headers: {
    'Content-Type': 'application/json'
  },
  body: JSON.stringify({
    nombre: "Ana",
    expediente: 456
  })
});

// Lado del SERVIDOR
app.use(express.json());
app.post('/api/estudiantes', (req, res) => {
  console.log(req.body.nombre);
});
```

Índice

1. Introducción
2. Entorno de Trabajo
3. Creación de Servidor
4. API servidor sobre MongoDB
5. API servidor sobre MySQL

Configuración de MySQL con Node.js[6]

```
import mysql from 'mysql2/promise';

// Configuración del Pool (mejor rendimiento que conexión única)
const db = mysql.createPool({
  host: 'localhost',
  user: 'root',      // Tu usuario de MySQL
  password: '',      // Tu contraseña
  database: 'clase', // Nombre de la BD
  waitForConnections: true,
  connectionLimit: 10
});
export default db;
```

- **mysql2/promise**: Permite usar la sintaxis `await` en las consultas.
- **createPool**: Gestiona múltiples conexiones simultáneas de forma eficiente, evitando que el servidor se bloquee.[5]

[5] El `export default db` se emplearía si trabajamos con varios archivos, usando luego `import db from './config/db.js'`

[6] Lo instalamos mediante: `npm install mysql2`

API de Estudiantes con MySQL (Lectura y Creación)

A diferencia de MongoDB, aquí enviamos sentencias SQL a través de la biblioteca `mysql2`.

```
// GET: Obtener todos los estudiantes
app.get('/api/estudiantes', async (req, res) => {
  const [rows] = await db.query('SELECT * FROM estudiantes');
  res.json(rows);
});

// POST: Registrar un nuevo estudiante (Uso de placeholders ?)
app.post('/api/estudiantes', async (req, res) => {
  const { nombre, expediente } = req.body;
  const sql = 'INSERT INTO estudiantes (nombre, expediente) VALUES (?, ?)';

  try {
    const [result] = await db.query(sql, [nombre, expediente]);
    res.status(201).json({ id: result.insertId, nombre, expediente });
  } catch (err) {
    res.status(500).json({ error: "Error al insertar" });
  }
});
```

Nótese que nunca concatenamos variables directamente en el SQL. Usamos el signo ? para evitar ataques de **Inyección SQL**.

API de Estudiantes con MySQL (Edición y Borrado)

```
// PUT: Actualizar datos por ID
app.put('/api/estudiantes/:id', async (req, res) => {
  const { id } = req.params;
  const { nombre } = req.body;
  const sql = 'UPDATE estudiantes SET nombre = ? WHERE id = ?';

  await db.query(sql, [nombre, id]);
    res.json({ mensaje: "Estudiante actualizado" });
  });

// DELETE: Eliminar un estudiante
app.delete('/api/estudiantes/:id', async (req, res) => {
  const { id } = req.params;
  await db.query('DELETE FROM estudiantes WHERE id = ?', [id]);
  res.status(204).send(); // Éxito, sin contenido
});
```

Diferencia Clave

En MySQL las tablas tienen un esquema rígido. Si intentas insertar un campo que no existe en la tabla, el servidor devolverá un error.

Comparativa de Persistencia en Node.js

Concepto	**MongoDB (NoSQL)**	**MySQL (Relacional)**
Estructura	Documentos JSON flexibles	Tablas con filas y columnas
Lenguaje	Métodos JS (`.find()`)	Sentencias SQL (`SELECT`)
Esquema	Dinámico (cambia fácil)	Rígido (debe definirse antes)
Identificador	`_id` (ObjectId)	`id` (Entero autoincremental)

Conclusión para el alumno

- Usa **MongoDB** si tus datos cambian mucho o no tienen una estructura fija.
- Usa **MySQL** si tienes datos muy relacionados y necesitas integridad total.